



1.4 — Variable assignment and initialization

👤 **ALEX¹** 🕒 **MARCH 22, 2024**

In the previous lesson ([1.3 -- Introduction to objects and variables](https://www.learncpp.com/cpp-tutorial/introduction-to-objects-and-variables/) (<https://www.learncpp.com/cpp-tutorial/introduction-to-objects-and-variables/>)²), we covered how to define a variable that we can use to store values. In this lesson, we'll explore how to actually put values into variables and use those values.

As a reminder, here's a short snippet that first allocates a single integer variable named *x*, then allocates two more integer variables named *y* and *z*:

```
1 | int x;    // define an integer variable named x
2 | int y, z; // define two integer variables, named y and z
```

Variable assignment

After a variable has been defined, you can give it a value (in a separate statement) using the `=` *operator*. This process is called **assignment**, and the `=` *operator* is called the **assignment operator**.

```
1 | int width; // define an integer variable named width
2 | width = 5; // assignment of value 5 into variable width
3 |
4 | // variable width now has value 5
```

By default, assignment copies the value on the right-hand side of the `=` *operator* to the variable on the left-hand side of the operator. This is called **copy assignment**.

Here's an example where we use assignment twice:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int width;
6 |     width = 5; // copy assignment of value 5 into variable width
7 |
8 |     std::cout << width; // prints 5
9 |
10 |    width = 7; // change value stored in variable width to 7
11 |
12 |    std::cout << width; // prints 7
13 |
14 |    return 0;
15 | }
```

This prints:

57

When we assign value 7 to variable *width*, the value 5 that was there previously is overwritten. Normal variables can only hold one value at a time.

Warning

One of the most common mistakes that new programmers make is to confuse the assignment operator (`=`) with the equality operator (`==`). Assignment (`=`) is used to assign a value to a variable. Equality (`==`) is used to test whether two operands are equal in value.

Initialization

One downside of assignment is that it requires at least two statements: one to define the variable, and another to assign the value.

These two steps can be combined. When an object is defined, you can optionally give it an initial value. The process of specifying an initial value for an object is called **initialization**, and the syntax used to initialize an object is called an **initializer**.

```
1 | int width { 5 }; // define variable width and initialize with initial value
2 | 5
3 |
   | // variable width now has value 5
```

In the above initialization of variable `width`, `{ 5 }` is the initializer, and `5` is the initial value.

Different forms of initialization

Initialization in C++ is surprisingly complex, so we'll present a simplified view here.

There are 6 basic ways to initialize variables in C++:

```
1 | int a;           // no initializer (default initialization)
2 | int b = 5;       // initial value after equals sign (copy initialization)
3 | int c( 6 );      // initial value in parenthesis (direct initialization)
4 |
5 | // List initialization methods (C++11) (preferred)
6 | int d { 7 };     // initial value in braces (direct list initialization)
7 | int e = { 8 };   // initial value in braces after equals sign (copy list
8 | initialization)
   | int f {};        // initializer is empty braces (value initialization)
```

You may see the above forms written with different spacing (e.g. `int d{7};`). Whether you use extra spaces for readability or not is a matter of personal preference.

Default initialization

When no initializer is provided (such as for variable `a` above), this is called **default initialization**. In most cases, default initialization performs no initialization, and leaves a variable with an indeterminate value.

We'll discuss this case further in lesson ([1.6 -- Uninitialized variables and undefined behavior](https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/) (<https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/>)³).

Copy initialization

When an initial value is provided after an equals sign, this is called **copy initialization**. This form of initialization was inherited from C.

```
1 | int width = 5; // copy initialization of value 5 into variable width
```

Much like copy assignment, this copies the value on the right-hand side of the equals into the variable being created on the left-hand side. In the above snippet, variable `width` will be initialized with value `5`.

Copy initialization had fallen out of favor in modern C++ due to being less efficient than other forms of initialization for some complex types. However, C++17 remedied the bulk of these issues, and copy initialization is now finding new advocates. You will also find it used in older code (especially code ported from C), or by developers who simply think it looks more natural and is easier to read.

For advanced readers

Copy initialization is also used whenever values are implicitly copied or converted, such as when passing arguments to a function by value, returning from a function by value, or catching exceptions by value.

Direct initialization

When an initial value is provided inside parenthesis, this is called **direct initialization**.

```
1 | int width( 5 ); // direct initialization of value 5 into variable width
```

Direct initialization was initially introduced to allow for more efficient initialization of complex objects (those with class types, which we'll cover in a future chapter). Just like copy initialization, direct initialization had fallen out of favor in modern C++, largely due to being superseded by list initialization. However, we now know that list initialization has a few quirks of its own, and so direct initialization is once again finding use in certain cases.

For advanced readers

Direct initialization is also used when values are explicitly cast to another type.

One of the reasons direct initialization had fallen out of favor is because it makes it hard to differentiate variables from functions. For example:

```
1 | int x(); // forward declaration of function x
2 | int x(0); // definition of variable x with initializer 0
```

List initialization

The modern way to initialize objects in C++ is to use a form of initialization that makes use of curly braces. This is called **list initialization** (or **uniform initialization** or **brace initialization**).

List initialization comes in three forms:

```
1 | int width { 5 }; // direct list initialization of initial value 5 into
2 | variable width
3 | int height = { 6 }; // copy list initialization of initial value 6 into
   | variable height
   | int depth {}; // value initialization (see next section)
```

As an aside...

Prior to the introduction of list initialization, some types of initialization required using copy initialization, and other types of initialization required using direct initialization. List initialization was introduced to provide a more consistent initialization syntax (which is why it is sometimes called “uniform initialization”) that works in most cases.

Additionally, list initialization provides a way to initialize objects with a list of values (which is why it is called “list initialization”). We show an example of this in lesson [16.2 -- Introduction to std::vector and list constructors](https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/) (<https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/>)⁴.

List initialization has an added benefit: “narrowing conversions” in list initialization are ill-formed. This means that if you try to brace initialize a variable using a value that the variable can not safely hold, the compiler is required to produce a diagnostic (usually an error). For example:

```
1 | int width { 4.5 }; // error: a number with a fractional value can't fit into
   | an int
```

In the above snippet, we’re trying to assign a number (4.5) that has a fractional part (the .5 part) to an integer variable (which can only hold numbers without fractional parts).

Copy and direct initialization would simply drop the fractional part, resulting in the initialization of value 4 into variable *width*. Your compiler may optionally warn you about this, since losing data is rarely desired. However, with list initialization, your compiler is required to generate a diagnostic in such cases.

Conversions that can be done without potential data loss are allowed.

To summarize, list initialization is generally preferred over the other initialization forms because it works in most cases (and is therefore most consistent), it disallows narrowing conversions, and it supports initialization with lists of values (something we'll cover in a future lesson). While you are learning, we recommend sticking with list initialization (or value initialization).

Best practice

Prefer direct list initialization (or value initialization) for initializing your variables.

Author's note

Bjarne Stroustrup (creator of C++) and Herb Sutter (C++ expert) also recommend [using list initialization](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-list) (<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-list>)⁵ to initialize your variables.

In modern C++, there are some cases where list initialization does not work as expected. We cover one such case in lesson [16.2 -- Introduction to std::vector and list constructors](https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/) (<https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/>)⁴.

Because of such quirks, some experienced developers now advocate for using a mix of copy, direct, and list initialization, depending on the circumstance. Once you are familiar enough with the language to understand the nuances of each initialization type and the reasoning behind such recommendations, you can evaluate on your own whether you find these arguments persuasive.

Value initialization and zero initialization

When a variable is initialized using empty braces, **value initialization** takes place. In most cases, **value initialization** will initialize the variable to zero (or empty, if that's more appropriate for a given type). In such cases where zeroing occurs, this is called **zero initialization**.

```
1 | int width {}; // value initialization / zero initialization to value 0
```

Q: When should I initialize with { 0 } vs {}?

Use an explicit initialization value if you're actually using that value.

```
1 | int x { 0 }; // explicit initialization to value 0
2 | std::cout << x; // we're using that zero value
```

Use value initialization if the value is temporary and will be replaced.

```
1 | int x {}; // value initialization
2 | std::cin >> x; // we're immediately replacing that value
```

Initialize your variables

Initialize your variables upon creation. You may eventually find cases where you want to ignore this advice for a specific reason (e.g. a performance critical section of code that uses a lot of variables), and that's okay, as long the choice is made deliberately.

Related content

For more discussion on this topic, Bjarne Stroustrup (creator of C++) and Herb Sutter (C++ expert) make this recommendation themselves [here](https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es20-always-initialize-an-object) (<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es20-always-initialize-an-object>)⁶.

We explore what happens if you try to use a variable that doesn't have a well-defined value in lesson [1.6 -- Uninitialized variables and undefined behavior](https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/) (<https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/>)³.

Best practice

Initialize your variables upon creation.

Initializing multiple variables

In the last section, we noted that it is possible to define multiple variables *of the same type* in a single statement by separating the names with a comma:

```
1 | int a, b;
```

We also noted that best practice is to avoid this syntax altogether. However, since you may encounter other code that uses this style, it's still useful to talk a little bit more about it, if for no other reason than to reinforce some of the reasons you should be avoiding it.

You can initialize multiple variables defined on the same line:

```
1 | int a = 5, b = 6;           // copy initialization
2 | int c( 7 ), d( 8 );        // direct initialization
3 | int e { 9 }, f { 10 };     // direct brace initialization
4 | int g = { 9 }, h = { 10 }; // copy brace initialization
5 | int i {}, j {};           // value initialization
```

Unfortunately, there's a common pitfall here that can occur when the programmer mistakenly tries to initialize both variables by using one initialization statement:

```
1 | int a, b = 5; // wrong (a is not initialized!)
2 |
3 | int a = 5, b = 5; // correct
```

In the top statement, variable “a” will be left uninitialized, and the compiler may or may not complain. If it doesn’t, this is a great way to have your program intermittently crash or produce sporadic results. We’ll talk more about what happens if you use uninitialized variables shortly.

The best way to remember that this is wrong is to consider the case of direct initialization or brace initialization:

```
1 | int a, b( 5 );  
2 | int c, d{ 5 };
```

Because the parenthesis or braces are typically placed right next to the variable name, this makes it seem a little more clear that the value 5 is only being used to initialize variable *b* and *d*, not *a* or *c*.

Unused initialized variables warnings

Modern compilers will typically generate warnings if a variable is initialized but not used (since this is rarely desirable). And if “treat warnings as errors” is enabled, these warnings will be promoted to errors and cause the compilation to fail.

Consider the following innocent looking program:

```
1 | int main()  
2 | {  
3 |     int x { 5 }; // variable defined  
4 |  
5 |     // but not used anywhere  
6 |  
7 |     return 0;  
8 | }
```

When compiling this with the g++ compiler, the following error is generated:

```
prog.cc: In function 'int main()':  
prog.cc:3:9: error: unused variable 'x' [-Werror=unused-variable]
```

and the program fails to compile.

There are a few easy ways to fix this.

1. If the variable really is unused, then the easiest option is to remove the definition of `x` (or comment it out). After all, if it’s not used, then removing it won’t affect anything.
2. Another option is to simply use the variable somewhere:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int x { 5 };
6 |
7 |     std::cout << x; // variable now used somewhere
8 |
9 |     return 0;
10 | }
```

But this requires some effort to write code that uses it, and has the downside of potentially changing your program's behavior.

The `[[maybe_unused]]` attribute C++17

In some cases, neither of the above options are desirable. Consider the case where we have a bunch of math/physics values that we use in many different programs:

```
1 | int main()
2 | {
3 |     double pi { 3.14159 };
4 |     double gravity { 9.8 };
5 |     double phi { 1.61803 };
6 |
7 |     // assume some of the above are used here, some are not
8 |
9 |     return 0;
10 | }
```

If we use these a lot, we probably have these saved somewhere and copy/paste/import them all together.

However, in any program where we don't use *all* of these values, the compiler will complain about each variable that isn't actually used. While we could go through and remove/comment out the unused ones for each program, this takes time and energy. And later if we need one that we've previously removed, we'll have to go back and re-add it.

To address such cases, C++17 introduced the `[[maybe_unused]]` attribute, which allows us to tell the compiler that we're okay with a variable being unused. The compiler will not generate unused variable warnings for such variables.

The following program should generate no warnings/errors:


```
1 | int main()
2 | {
3 |     [[maybe_unused]] double pi { 3.14159 };
4 |     [[maybe_unused]] double gravity { 9.8 };
5 |     [[maybe_unused]] double phi { 1.61803 };
6 |
7 |     // the above variables will not generate unused variable warnings
8 |
9 |     return 0;
10| }
```

Additionally, the compiler will likely optimize these variables out of the program, so they have no performance impact.

Author's note

In future lessons, we'll often define variables we don't use again, in order to demonstrate certain concepts. Making use of `[[maybe_unused]]` allows us to do so without compilation warnings/errors.

Quiz time

Question #1

What is the difference between initialization and assignment?

[Hide Solution \(javascript:void\(0\)\)](#)⁷

Initialization gives a variable an initial value at the point when it is created. Assignment gives a variable a value at some point after the variable is created.

Question #2

What form of initialization should you prefer when you want to initialize a variable with a specific value?

[Hide Solution \(javascript:void\(0\)\)](#)⁷

Direct list initialization (aka. direct brace initialization).

Question #3

What are default initialization and value initialization? What is the behavior of each? Which should you prefer?

[Hide Solution \(javascript:void\(0\)\)](#)⁷

Default initialization is when a variable initialization has no initializer (e.g. `int x;`). In most cases, the variable is left with an indeterminate value.

Value initialization is when a variable initialization has an empty brace (e.g. `int x{};`). In most cases this will perform zero-initialization.

You should prefer value initialization to default initialization.



Next lesson

1.5 [Introduction to iostream: cout, cin, and endl](#)

8



Back to table of contents

9



Previous lesson

1.3 [Introduction to objects and variables](#)

2

10



B **U** **URL** **INLINE CODE** **C++ CODE BLOCK** **HELP!**

Leave a comment...



 Name*


@ Email* | 

Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above! 

 Avatars from <https://gravatar.com/>¹² are connected to your provided email address.

452 COMMENTS

Newest ▼



Cyber

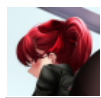
🕒 March 29, 2024 9:41 pm

As an experienced developer myself, no way you are giving such quality education for free. I especially love the best practices part in every section, it helps so much in building my own

opinions about the language itself.

Expect a donation soon!

👍 4 ➡ Reply

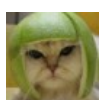


William

🕒 March 28, 2024 8:31 pm

What is the difference between Modern C++, Old C Style C++, and Java Style C++? Is the modern way of initializing variables using the {}?

👍 0 ➡ Reply



Alex Author

➡ Reply to [William](#)¹³ 🕒 March 29, 2024 4:40 pm

Yes, the modern way is using list initialization.

👍 1 ➡ Reply



habibi

🕒 March 26, 2024 5:21 pm

This is like a book without the overhead of an actual book and is engaging, I would honestly pay you, any buymeacoffees?

👍 1 ➡ Reply



Cope

➡ Reply to [habibi](#)¹⁴ 🕒 March 28, 2024 6:52 am

Agreed. I would strongly recommend a donation feature; go the Wikipedia route.

👍 0 ➡ Reply



Alex Author

➡ Reply to [Cope](#)¹⁵ 🕒 March 29, 2024 4:21 pm

FWIW, there is a donate link under the About menu. Thanks for your support!

👍 0 ➡ Reply



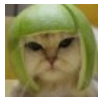
dumi

🕒 March 22, 2024 3:08 pm

I don't get why direct list initialization is preferred. I always see C++ programmers initializing values with the copy initialization method (`int a=5`) and almost never see direct list initialization (`int a{5}`)

👍 0

➡ Reply



Alex Author

👤 Reply to [dumi](#)¹⁶ ⌚ March 22, 2024 5:02 pm

From the lesson: "List initialization is generally preferred over the other initialization forms because it works in most cases (and is therefore most consistent), it disallows narrowing conversions, and it supports initialization with lists of values".

I don't know what kind of code you're looking at so I can't comment on why you're mostly seeing copy init used.

👍 0

➡ Reply



Bob

⌚ March 21, 2024 10:09 am

'maybe_unused' attribute directive ignored [-Werror=attributes]
I am getting this error, please help.

👍 0

➡ Reply



Alex Author

👤 Reply to [Bob](#)¹⁷ ⌚ March 21, 2024 3:04 pm

If you're using C++14, you might get this error. Either upgrade to C++17 (or newer) or remove the `[[maybe_unused]]`.

👍 0

➡ Reply



Forest Wang

⌚ February 26, 2024 12:01 pm

default initialization means uninitialized. Am I right?

👍 0

➡ Reply



Alex Author

👤 Reply to [Forest Wang](#)¹⁸ ⌚ February 27, 2024 8:29 pm

Somewhat. For fundamental types, default initialization means uninitialized. For some other types (class types, which we cover in a future lesson) this is not necessarily the case.



3



Reply

**Forest Wang**

🕒 February 26, 2024 11:55 am

```
int a;  
int b();  
int c(0);  
int d{};  
int e{0};
```

What is the difference? Please elaborate.



0



Reply

**Alex**

Author

Reply to [Forest Wang](#)¹⁹ 🕒 February 27, 2024 8:29 pm

This sounds suspiciously like a homework problem. All of these are answered on this lesson or lesson 1.6 (except b, which is a function prototype).



11



Reply

**Grizz**

🕒 February 24, 2024 5:25 pm

I'm confused if I should be putting code examples in and run them to see if I get the same results. I tried inputting `int x { 5 };` and other variations but the same error saying its initialized but not referenced keeps coming up. I assume that referencing is in future lessons but I just wasn't sure if I'm just wasting my time at the moment trying to recreate the examples or just hold off on that until the lessons instructs me to do so



0



Reply

**Alex**

Author

Reply to [Grizz](#)²⁰ 🕒 February 25, 2024 8:52 pm

Generally any example with a `main()` function is meant to be compiled. The rest are snippets showing something specific.

Some examples may intentionally not compile.



0



Reply

**CoolCoder09**

🕒 February 15, 2024 2:37 am

So for the `[[maybe_unused]]` attribute, should we use that attribute every time we initialize a variable? or should we just put it in for variables that won't be used in the code?

👍 0 ➡ Reply



Alex Author

👤 Reply to [CoolCoder09](#)²¹ ⌚ February 16, 2024 11:03 am

Only for variables that we reasonably expect may not be used. This typically occurs when we have a set of related variables that we want to keep together for organizational purposes, but for which only some may actually be used by a given program.

👍 2 ➡ Reply



IceFloe

⌚ February 14, 2024 7:50 pm

How not to get confused by the fact that direct initialization (with parentheses) is similar to the declaration of the function `int main()` and `double weight (5.4)`?

And secondly, can I declare other variables inside variables, for example: `int a {int b {3}, c {4}}`?

Thanks for the lesson, I hope direct initialization of the list will be enough, unless we want to assign a different value to the same variable using the assignment operator `=)))`

🔗 Last edited 1 month ago by IceFloe

👍 0 ➡ Reply



Alex Author

👤 Reply to [IceFloe](#)²² ⌚ February 16, 2024 10:53 am

1. This is one of the main reasons parenthesis initialization fell out of favor. You can tell the difference because the function declaration has zero or more type parameters, whereas a variable definition using parenthesis initialization will have one or more value parameters.
2. No. Variable definitions are a statement. Variable initializers are an expression.
3. Direct list initialization works in the vast majority of cases. We cover the one common case where it doesn't in a future lesson (<https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/>)

👍 0 ➡ Reply



IceFloe

👤 Reply to [Alex](#)²³ ⌚ February 16, 2024 2:45 pm

Thanks))

👍 0

➡ Reply



Mohamed

🕒 February 12, 2024 7:41 am

- you said: In most cases, default initialization leaves a variable with an indeterminate value.
- in most cases ?!!!!
- are there other cases in which default initialization can leave a variable with determinate value ?

🔗 Last edited 1 month ago by Mohamed

👍 0

➡ Reply



Alex Author

👤 Reply to [Mohamed](#)²⁴ 🕒 February 15, 2024 3:16 pm

Yes. Variables with static duration are zero initialized, and objects that have a class type can self-initialize to whatever values they want. These are both covered in future lessons, when we introduce static duration and class types respectively.

👍 0

➡ Reply



snooty

🕒 February 2, 2024 6:20 pm

why should we prefer value initialization over default initialization? Isn't an indeterminate value better than it being zero when you don't want it to?

👍 0

➡ Reply



Alex Author

👤 Reply to [snooty](#)²⁵ 🕒 February 3, 2024 11:41 am

We prefer all of our objects have known, fixed values, so that our program is more likely to behave consistently if we have an error. That makes said errors easier to track down.

👍 4

➡ Reply



Riaet

🕒 January 27, 2024 12:39 am

Hello there,

```
int width { 5 }; // direct list initialization of value 5 into variable width
```

```
int height = { 6 }; // copy list initialization of value 6 into variable height
```

is there any difference between these two?

 Last edited 2 months ago by [Riaet](#)

 0

 Reply



Alex Author

 Reply to [Riaet](#)²⁶  January 28, 2024 4:50 pm


In this particular case, no. But you should generally prefer the former.

 0

 Reply



naiven

 January 25, 2024 4:03 am

hello, i hope you're doing well, im liking the tutorial so far and would like to ask what problem is in the following code if possible? it keeps returning only the else statement no matter what i write and can't seem to see any errors, also i configured visual studio as per the tutorial following and am on c++20 on a windows *64.

ps: i don't know how to put it in code like sorry

```
#include <iostream>
```

```
using namespace std;
```

```
//stocks data about products on sell
```

```
int main()
```

```
{
```

```
int panda_biscuit = 1, cacao = 2;
```

```
double toy = 3;
```

```
int uiSearch;
```

```
cout << "which product do you want ?" << endl;
```

```
cin >> uiSearch;
```

```
// how to put the quantity in disposition ?
```

```
//list the diponibility
```

```
if (uiSearch == 1) {
```

```
cout << "out of stock" << endl;
```

```
}
```

```
else if (uiSearch == 2){
```

```
cout << "in stock" << endl;
```



```
}  
else if (uiSearch == 3){  
    cout << "in store only" << endl;  
}  
else {  
    cout << "product unrecognized" << endl;  
}  
return 0;  
}
```

 Last edited 2 months ago by naiven

 0  Reply



stevelearningcpp

 Reply to [naiven](#)²⁷  January 25, 2024 3:16 pm

Try printing the value of uiSearch.

 0  Reply



naiven


 Reply to [stevelearningcpp](#)²⁸  January 30, 2024 10:57 am

a lil late but thanks tho

 0  Reply



naiven

 Reply to [stevelearningcpp](#)²⁸  January 27, 2024 10:18 am

it return 0 like it's uninitialized when i did it (i think), oh i just tried something like putting numbers instead of text in the console and it works.

but how you put text if i put:

```
string uiSearch ;  
if (uiSearch == 1) {  
    cout<< "out of stock<< endl  
}
```

it wouldn't work, but if i don't initialize the products, it will result in an error

 Last edited 2 months ago by naiven

 0  Reply

**Leonid**

🕒 January 9, 2024 2:49 pm

Last time I've been studying C++ was in 2003-2006, many things have changed since then. Thanks for your efforts on this site, updating my knowledge and learning new things.

👍 2 ➡ Reply

**minuteman217**

🕒 December 17, 2023 8:44 am

Learning a bunch from these tutorials, thanks Alex! Thank the Lord I switched to learning from this site rather than w3schools, or I would have ingrained many bad habits into my brain.

👍 5 ➡ Reply

**AbeerOrTwo**

🕒 December 13, 2023 7:55 pm

Learned something new, been using copy initialization but switching over to list initialization.

👍 2 ➡ Reply

**Tim**

🕒 December 10, 2023 3:35 am

Hi, I started learning C++ and your tutorial is very helpful.

Still I got a question, maybe someone can redirect me to another lesson or explain.

In the code below, when to prefer which initialization?

After having read the article, for `int a` direct list initialization is best.

What is the difference (in terms of what the compiler does) between initialization for `int b` and `int c`? Which one to use in which case?

Probably I come up with that question because now there are variables on the right-hand side instead of literals.

```

1 | int main() {
2 |
3 |     int a {1}; // direct list initialization
4 |     int b {a}; // Q: how does b get its value?
5 |     int c = a; // copy initialization
6 |
7 |     a += 4;
8 |
9 |
10 |     std::cout << a << ", " << b << " " << c << std::endl;
11 |     return 0;
12 | }

```

Output: 5, 1, 1 (as expected).

👍 0 ➡ Reply



Alex Author

🗨 Reply to [Tim](#)²⁹ 🕒 December 11, 2023 11:56 am

It's best to use direct list initialization for all three cases. In the case of `b`, the variable `a` will be evaluated to get its value (1), which will then be used to initialize `b`. Same with `c`.

With fundamental types, the compiler essentially does the same thing for copy, direct, and list initialization. But with list initialization, narrowing conversions are an error, which is useful.

👍 3 ➡ Reply



Old Camel

🕒 December 5, 2023 6:29 am

Thanks to Alex and the team, I learned a lot from this website.

👍 0 ➡ Reply



Damian

🕒 November 30, 2023 2:28 am

>> Initialize your variables upon creation.

I might be missing something here, but how would you create a variable without initializing it? The cppcoreguidelines link in the block above it confuses me even more, because they say:

```

1 | int i; // bad: uninitialized variable

```

but then say:

```
1 | string s; // OK: default initialized
```

but wouldn't the first block also be default initialized?

👍 0

➡ Reply



Bailiwick

🗨 Reply to [Damian](#)³⁰ 🕒 December 1, 2023 1:17 am

```
1 | int i; // Here, at runtime OS will allocate required amount of
2 | memory to variable named i.
3 |     // Suppose, if variable i is not used in entire program,
4 | then compiler will issue a warning and it won't write
5 |     // instruction that allocate memory to i. It will simply get
6 | ignored by compiler.
7 |     // That's why creation of variable actually means allocation
8 | of memory to the variable ( instantiation ).
9 |     // Uninitialized variable simply means it has not been given
10 | a value so if variable is created at runtime then
11 |     // it will just have a garbage value. ( it will be created
12 | even if it used uninitialized though with a warning ) .
13 |     // Some compiler may put a value for initializing 'i' which
14 | depends entirely on compiler. But we should not depend
15 | // on compiler to initialize it.
16 |
17 | string s; // Here s is a object of type string, which has it own
18 | default "constructor" which initializes s to empty string
19 | // implicitly.
```

✎ Last edited 4 months ago by Bailiwick

👍 0

➡ Reply



Alex Author

🗨 Reply to [Bailiwick](#)³¹ 🕒 December 1, 2023 2:19 pm

`int i` creates an uninitialized variable. Technically `i` is default-initialized, but default initialization leaves fundamental types uninitialized, so we typically just say that `i` is not initialized.

Class types (of which `std::string` is one) can self-initialize, so technically using default initialization with a class type is okay. However, using value-initialization is preferred, as it behaves more predictably.

👍 0

➡ Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/introduction-to-objects-and-variables/>
3. <https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/>
4. <https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/>
5. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-list>
6. <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es20-always-initialize-an-object>
7. `javascript:void(0)`
8. <https://www.learncpp.com/cpp-tutorial/introduction-to-iostream-cout-cin-and-endl/>
9. <https://www.learncpp.com/>
10. <https://www.learncpp.com/variable-assignment-and-initialization/>
11. <https://www.learncpp.com/cpp-tutorial/configuring-your-compiler-warning-and-error-levels/>
12. <https://gravatar.com/>
13. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-595206>
14. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-595148>
15. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-595186>
16. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-594988>
17. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-594924>
18. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-594061>
19. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-594060>
20. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-594007>
21. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-593646>
22. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-593641>
23. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-593727>
24. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-593594>
25. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-593179>
26. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-592938>
27. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-592848>
28. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-592880>
29. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-590728>
30. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-590383>
31. <https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/#comment-590414>